Real-Valued Genetic Optimization

Intro to Engineering Optimization Final Project

Joshua Andrew Haas
Electrical and Computer Engineering
Rowan University
Glassboro, NJ, USA
haasj74@students.rowan.edu

***Abstract*—Recently, numerous so-called "meta-heuristic" algorithms have been proposed to solve optimization problems. Most of these find inspiration from nature rather than using a mathematically devised process. One example is the idea of a genetic algorithm, also known as evolutionary optimization, that attempts to solve optimization problems by modeling the processes driving evolution, namely mutation and natural selection. These meta-heuristic algorithms, including genetic algorithms, hope to find the global optimum (given enough iterations), rather than the much-dreaded local optimum. However, the classic genetic algorithm is comprised entirely of binary variables, and is thus intrinsically discrete or categorical in nature. It has been used, for example, to solve the traveling salesman problem. It is possible to extend genetic optimization algorithms to the continuous case, where they are called "real-valued" genetic algorithms. In this case each variable is continuous and can take on an infinite number of different values. After a brief literature review in the field of genetic algorithms, this paper examines the creation of a real-valued genetic algorithm in the Python programming language and its performance on numerous stock optimization functions.**

Keywords—optimization, genetic algorithm, meta-heuristic, global minimum, python, convergence

I. Introduction

At its simplest, optimization is applying a technique or algorithm to find the minimum or maximum of some cost function. For a very easy example, such an algorithm would locate the point (0,0) with a value of 0 as the minimum of the function $f(x) = x^2$. One of the easiest algorithms to understand is the Steepest Descent algorithm, which works by calculating the derivative of the cost function. First, a random starting point is chosen somewhere in the function's valid input space, for example at $x = 1$. Then, the gradient (multi-dimensional derivative) of the function at that point is calculated. In our 1D example, the gradient is the derivative, and at $x = 1$ the derivative is $f'(x) = 2x = 2(1) = 2$. The algorithm then moves in the opposite direction of the gradient. In our example, since the gradient is positive, the algorithm moves in the negative x direction, thus getting closer to the minimum at $x = 0$. The process of taking the derivative and moving the current point is repeated until some stopping condition is met, for example when the decrease in the cost function from one iteration to the next is less than a threshold value. The update rule for the Steepest Descent algorithm is shown below as Eq. 1, where x_{k+1} is the next value, x_k is the previous value, α_k is a variable step size, ∇ is the gradient, and $f(x_k)$ is the cost function evaluated at x_k [1].

$$x_{(k+1)} = x_k - \alpha_k \nabla f(x_k) \qquad \text{Eq. 1}$$

However, one of the major shortcomings of this algorithm (and other gradient-based algorithms) is its tendency to converge to local minima. For example, the function presented below as Eq. 2 and shown in Fig. 1 has a global minimum at $x = 0$, but it also has local minima at $x = \pm 2\pi$. If the Steepest Descent algorithm began at, say, $x = 5$ it would very likely converge to the local minimum at $x = 2\pi$ since the gradient at $x = 5$ is negative.

$$f(x) = |x| - 2\cos(x) \qquad \text{Eq. 2}$$

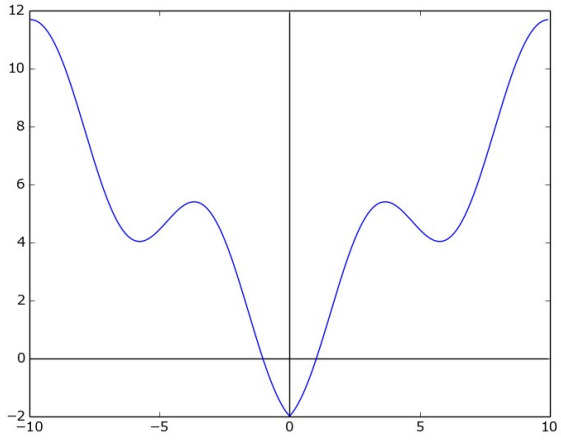


Figure 1. Graph of Eq. 2 showing local minima.

A related problem to that of local minima is functions with large search spaces. For example, in the Traveling Salesman Problem, a salesman wants to visit a number of cities while traveling the shortest distance possible and visiting each city

only once. The locations of each city are known, and result in different distances between each pair of cities. The only way to find the true global minimum is to evaluate every possible permutation. However, this becomes computationally infeasible very quickly, since the number of permutations increases factorially. For example, for 20 cities there are 121.6 quadrillion permutations. This large of a search space likely contains countless local minima that traditional optimization techniques would get stuck in.

One solution to the problem of local minima and large search spaces are stochastic approaches. In these methods some degree of randomness is introduced in the hopes of randomly hitting the global minimum (or at least getting close to it). Many of these stochastic approaches are based on phenomena observed in nature, giving rise to so-called biological algorithms. One such algorithm is the genetic algorithm, which is based on the theory of evolution. It should be noted that few if any of these algorithms have any mathematical backing or proof. Usually they "just work."

Evolution consists of two key concepts, natural selection and mutation. Natural selection is the process by which more "fit" individuals in a population are more likely to spread their genes via reproduction. For example, a cat with a better sense of smell than other cats could be more likely to find food. Therefore, this cat is more likely to survive long enough to reproduce, and its children will likely inherit its improved sense of smell. Since these children are also more likely to survive than their peers, they are also more likely to reproduce, and after enough generations the majority of the population will have this better sense of smell.

The process of genetic recombination inherent in reproduction of most organisms is also a key factor in natural selection. Every trait is determined by a gene encoded in an organism's DNA. When two organisms reproduce, each of the child's genes is randomly selected from either the mother or father. This results in novel combinations of genes in each generation and increases the population's diversity.

Entirely new, never seen before, genes can be introduced to a population via mutation. Sometimes during the process of reproduction, instead of receiving the mother or father's version of a particular genes, the child instead receives a mutant version. This mutation is completely random and is completely unrelated to the genes of the mother or father. In this way, new genes are introduced to the population. A beneficial mutation will eventually spread to the entire population via natural selection, while a detrimental mutation will eventually die out by the same process.

The Simple Genetic Algorithm (SGA) is shown below as Algorithm 1. It models every individual as an array of binary numbers, where one or more of the numbers code a specific gene [2].

Algorithm 1. Simple Genetic Algorithm.

1. Generate a random population of individuals.
2. Evaluate the fitness of every individual.
3. Breed (crossover and mutate) the most fit individuals from the current population to produce offspring.
4. Replace the current population with their offspring.
5. Repeat steps 2-4 until some stopping criteria.

In step 1, every individual is initialized as a random sequence of 1s and 0s. In step 2, the fitness function generates some fitness value given an individual's binary representation. Oftentimes the binary representation must first be decoded into a more meaningful set of numbers before the fitness function can be applied. In step 3, two individuals are randomly selected from the most fit individuals in the population (for example, from the top 50%). These two individuals (referred to as the mother and father) are then bred to create a new individual (referred to as the child). First, the genes from the mother and father undergo crossover (although there are many different crossover schemes, we will only mention one here). A point is randomly chosen on the binary genetic array. Every gene to the left of the point is taken from the mother, and every gene to the right of the point is taken from the father. It should be noted that if the mother and father have the same value for a specific gene, then the child is guaranteed to have that value (barring mutation). Every gene is also subject to the possibility of a random mutation (simply flipping the bit) based on some mutation rate. In step 4, the current population is completely forgotten and replaced by the children bred in step 3. Steps 2-4 are then repeated until some stopping criteria is met. Each full iteration of the algorithm is referred to as a generation [2].

Continuous problems (such as finding the minimum of a continuous function like Eq. 2) can be modeled using the SGA. However, this requires modeling each variable as a series of binary numbers, resulting in quantization error. It also reduces the speed of the algorithm since the binary genes must first be converted into a decimal representation before the cost function can be computed [3].

Inherently continuous genetic algorithms, known as Real-Valued Genetic Algorithms, have been proposed to overcome this shortcoming of the SGA. In such algorithms, each gene (variable) is a continuous decimal value. This requires modified crossover and mutation methods, but otherwise functions the same as the SGA. These differences will be explored further in the Methods section [3].

II. Literature Review

A. Overview of Stopping Criteria

The stopping criteria of a genetic algorithm (actually any meta-heuristic algorithm) has traditionally been uncertain and variable. Perhaps the most common stopping criterion is to simply terminate after a preset number of generations. However, this is inaccurate at best and cannot guarantee convergence to a global minimum without knowing the shape of the fitness function. Another form of stopping criteria is the so-called genotypical stopping criterion, which stops the algorithm when the variance in the genes of the population has fallen below some threshold. Finally, there exist the so-called phenotypical stopping criterion, which stop the algorithm when the progress (i.e. reduction in the fitness function) is less than some threshold for some number of generations [4].

Markov Chains are a technique used to model the state space of a random process that depends only on the current state (i.e. memoryless). The transition from one state to every other possible state is assigned a probability. This modeling technique is perfect for evolutionary algorithms since the next generation only depends on the parents in the current generation. Using Markov Chains to model genetic algorithms, it can be shown that any population can be reached from any other population as long as the mutation rate is nonzero. This means that, as the number of generations increases, the probability of reaching a specific population, and thus the probability of visiting a specific individual, increase. Moreover, this guarantees that, given enough generations, the genetic algorithm will eventually find the global minimum. While this guarantees eventual convergence, without further analysis, said convergence could potentially take longer than an exhaustive search [4].

B. Elitist Genetic Algorithms

The Elitist Genetic Algorithm (EGA) introduces a slight modification to the SGA. In this version, the children must compete with their parents in order to be part of the new population. In other words, the current generation is not simply forgotten and replaced by its children. Instead, the children only replace parents if the children are more fit than the parents. This guarantees that the algorithm never backtracks to a worse fitness value, since it will always remember its current best individual. In addition, good solutions are never lost during the search process as may be the case with the SGA. The elitist approach is also less sensitive to undersized populations, and lends itself well to hybridization with gradient-descent methods [5].

In the elitist version, using Markov Chains, an upper bound for the number of generations required to guarantee that all individuals have been visited can be computed. This bound is presented below as Eq. 3, where t is the number generations, α is the probability of having visited every individual, n is the population size, μ is the mutation rate, and l is the length of the chain representing an individual [4].

$$t \geq \left\lceil \frac{\ln(1-\alpha)}{n \ln(1-min\{\mu^l,(1-\mu)^l\})} \right\rceil \qquad \text{Eq. 3}$$

Unfortunately the above equation only applies in the case of the classic (i.e. discrete or binary) SGA. For continuous elitist genetic algorithms, Bhandari et al. used the variance of individuals to define a stopping criterion. The algorithm must first remember the most fit individual for the past n generations. Then the variance of these fitness values is calculated and compared to a threshold. If it is below the threshold then the algorithm stops. However, the authors note that the parameter n still depends on the shape of the fitness function. The calculated variance is shown below in Fig. 2, to proves that for most problems it does asymptotically approach zero, lending itself well to a convergence criterion [6].

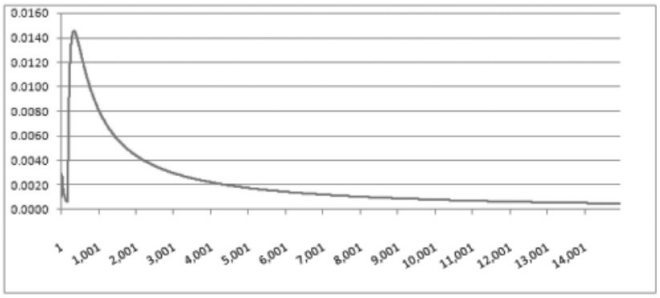


Figure 2. Variance of past best versus generation.

C. Adaptive Crossover and Mutation Rates

The population size, crossover rate, and mutation rate are the three most important parameters in most genetic algorithms. However, choosing optimum or even "good enough" values is highly problem dependent and not well understood at this time. Therefore, Lin at al. sought to develop methods for adaptive crossover and mutation rates. In addition, they suggest starting the algorithm with a very high mutation rate of 0.5 to help promote searching.

During the breeding step in each generation, the fitness values of every pair of parents and their respective offspring are compared. If on average the parents have a higher fitness value, then crossover has resulted in a decrease in fitness and the crossover rate is lowered. If on average the children have a higher fitness value, then crossover has resulted in an increase in fitness and the crossover rate is increased. This is also preformed to compare the children before and after the mutation step. If pre-mutation children have, on average, higher fitness values, then mutation has decreased the fitness of the population and the mutation rate is decreased. If post-mutation children have, on average, higher fitness values, then mutation has increased the fitness of the population and the mutation rate is increased. Both the crossover and mutation rate are capped at minimum and maximum values so both are always higher than zero and lower than one. This scheme outperformed existing crossover and mutation selection techniques on several optimization problems [2].

D. Differential Crossover

Although genetic algorithms were inspired by the process of evolution, nothing requires that their design be restricted to the exact parameters of real-life human evolution. One such proposal seeks to inhibit premature convergence (i.e. local minimum) using so-called Differential Evolution (DE) crossover in continuous genetic algorithms. First, only a subset of the population is selected for breeding and potential replacement. For each selected individual, its replacement is calculated by the formula shown below as Eq. 4, where $ch_{ij}(t+1)$ is the "child," $ch_{ij}(t)$ is the "parent," F_1 and F_2 are random numbers in the range $(0,1)$, $ch_{pj}(t)$ and $ch_{qj}(t)$ are randomly selected individuals from the current population, and $ch_{bestj}(t)$ is the individual in the current population with the best fitness value [3].

$$ch_{ij}(t+1) = ch_{ij}(t) + F_1(ch_{pj}(t) - ch_{qj}(t)) + F_2(ch_{bestj}(t) - ch_{ij}(t)) \qquad \text{Eq. 4}$$

Each individual is only replaced if its replacement has a higher fitness value than itself, making this scheme elitist. Moreover, this technique does not have a crossover probability or mutation probability as in most other genetic algorithms. This DE crossover seeks to lessen premature convergence and thus force the algorithm to find a global minimum rather than a local minimum. The same authors then combine the DE process with traditional crossover and mutation to come up with their Simplified DE (SDE) technique. The update equation is shown below as Eq. 5, using the same symbols as Eq. 4 except for CR is the crossover rate and $ch_{rj}(t)$ is a third random individual from the current population.

$$ch_{ij}(t+1) = ch_{pj}(t) + CR\left(ch_{qj}(t) - ch_{rj}(t)\right) \quad \text{Eq. 5}$$

The results from the DE and SDE crossover schemes significantly outperformed SGA and EGA on many functions in the authors' tests. The SDE scheme is pictured in Fig. 3 below to emphasize that individuals are updated based on the mutual distance between population members.

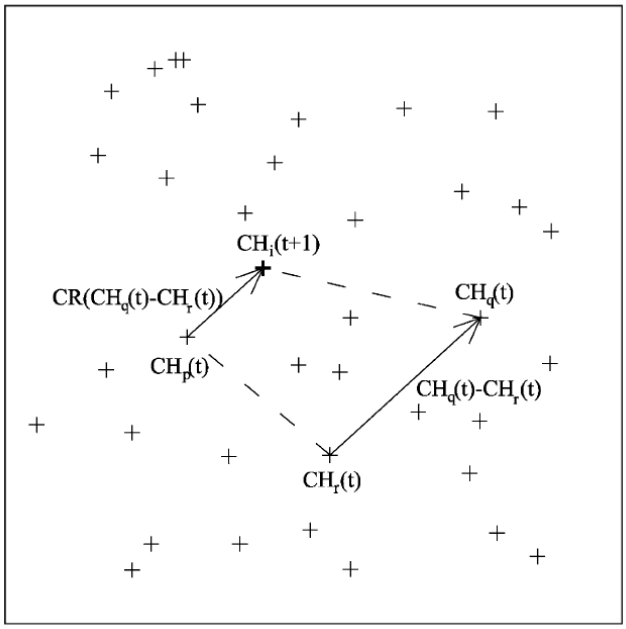


Figure 3. Geometric interpretation of SDE scheme.

E. Radioactive Zones

Another proposal to decrease the likelihood of local convergence is the introduction of so-called Radioactive Zones. Whenever the algorithm begins to stagnate, (i.e. population genes are barely changing), the algorithm creates a Radioactive Zone where the probability of mutation is 100%. The area of stagnation corresponds to a local minimum, so whenever an individual is created inside the radioactive zone it will immediately leave it via mutation. In addition, every time an individual is mutated by a radioactive zone, the mutation level in that zone is decreased. If the population stagnates for more generations than a threshold value, the algorithm stops. When adding this to the SDE crossover scheme described in the previous section, Hrtska et al. found that the resulting algorithm converged for all test functions, albeit often slower than competing algorithms [3].

F. Real-World Applications

1) Image Processing

Mashohor et al. used en EGA in their project for printed circuit board inspection. The goal of the project was to decide whether a series of circuit boards were acceptable or note by comparing a photograph of each board to a reference image. However, as boards are sent down the conveyor belt, they are almost always displaced or rotated relative to the center and orientation of the camera. Therefore, a simple pixel by pixel comparison with the reference image is impossible.

The authors thus implemented an EGA in order to estimate the displacement and rotation of the test board. According to previous work, genetic algorithms have produced fast and accurate matching while providing scaling and rotation consistency. The fitness function in this case was the similarity of the test and reference images pixel by pixel. Each individual was a 19-bit binary string, with 9 bits to represent rotation, 5 bits to represent x-axis displacement, and 5 bits to represent y-axis displacement. They created 200 generations with a population size of 18, crossover probability of 0.5, and mutation probability of 0.01. The displacements estimated by the GA were accurate to within a few pixels out of 1600 pixels, although the rotations estimated were significantly less accurate [7].

2) Power Distribution

Another potential application of genetic algorithms is power distribution optimization. For example, Tomoiaga et al. developed a modified genetic algorithm to find the distribution system with minimum active power loss. The genes in this application were binary, with each genes representing a specific link between two points on the graph of power distribution relays. The branch list of a sample graph is shown below in Fig. 4a [8].

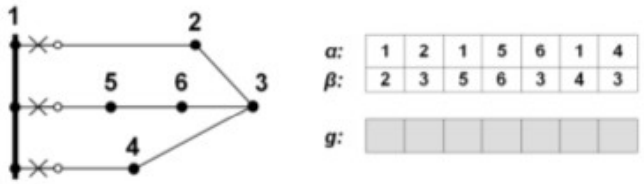


Figure 4a. Branch list of the graph.

The graph resulting from coding a sample population member is shown below in Fig. 4b. Applying the algorithm to several test problems, the authors found a reduction in power loss of up to 50% depending on the problem. This was achieved with only 10 population members and under a dozen generations using the C++ programming language. This algorithm performed significantly better in terms of both computation time and final result than traditional power distribution optimization algorithms.

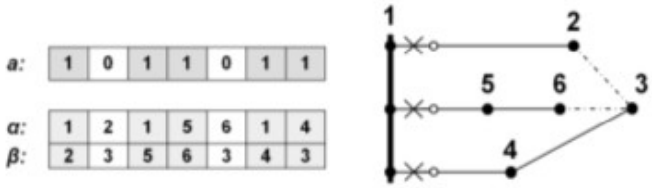


Figure 4b. Branch lists from decoding *a*.

3) Neural Network Training

Another very popular field in engineering and mathematics today is Pattern Recognition and Machine Learning. In a pattern recognition problem, the goal is to predict the classes of a set of observations, given their features, after training on a set of labeled data. For example, the classes could be male and female, and the features height and hair length. First, in training, the classifier (i.e. algorithm) is presented with a set of observations that note the height and hair length of random men and women. It uses these data to form some internal model. Next, in prediction, it is given a height and hair length, and outputs a prediction as to which class it "believes" that observation belongs, either male or female.

One popular classifier is the Neural Network, which consists of "neurons" that activate depending on the features. In order to train a neural network, however, requires optimizing the weights in each neuron based on the training data. Although this is traditionally done using a gradient descent algorithm, Ojha et al. instead used a genetic algorithm.

The authors sought to identify the presence of dangerous gases in Manholes in order to keep maintenance personnel safe. These gases can include Hydrogen Sulfide (H_2S), Ammonia (NH_3), and Methane (CH_4), all of which are extremely dangerous at significant concentrations. The easiest solution is to use an individual sensor for each dangerous gas type; however, most of these sensors also react to the presence of other gases. This gives an array of sensors significant cross-sensitivity that results in inaccurate results. Therefore, the authors seek to use a neural network classifier to improve prediction. Specifically, they used a real-valued genetic algorithm to optimize the weights for a two layer feed-forward neural network in the Java programming language. The performance met the minimum requirements, but was not compared to a neural network trained via gradient descent [9].

III. Methods

Since most meta-heuristic algorithms are intuitive rather than mathematically rigorous, I decided to implement a real-valued genetic algorithm on my own, and then compare my results with the recommendations in literature. I chose the Python programming language because I am very familiar with it and it is conducive to fast scripting of a wide variety of applications.

Two main files were created, namely "population.py" and "poptest.py". The "population.py" file contains a single class, namely "Population" that can be used to setup and execute a genetic algorithm optimization problem. During initialization, the user must specify a fitness function, constraint function, array of variables minima, and array of variable maxima. Optional parameters include providing a starting population, changing the population size, setting the kill rate, and setting the mutation rate. The algorithm is run by calling the "evolve" method on the created Population. The user can specify whether to check for convergence, the maximum number of iterations, and the tolerance for convergence. Note that the "scipy" package is required to run the algorithm [11]. These files can be found in Appendix A.

The algorithm implemented in "population.py" is shown below as Algorithm 2:

Algorithm 2

1. Generate a random population of individuals.
2. Evaluate the fitness of every individual.
3. Kill (remove) the least fit individuals based on the "kill_rate" parameter.
4. Breed (crossover and mutate) the remaining individuals from the current population to produce offspring to replace those killed.
5. Repeat steps 2-4 until standard deviation is less than some threshold.

The algorithm differs from other methods presented in the beginning of the paper in a few ways. First, the implementation of elitism is slightly different than in other papers. In most of the literature, elitism is implemented per family. In other words, during breeding, a mother and father produce two offspring. The two most fit individuals from this "family" of four are then chosen to enter the next population. In this implementation, the least fit half of the population is discarded. Then the remaining individuals are bred to repopulate and reach the desired population size.

In addition, the crossover method is different than that proposed in other literature. Given two individuals, suggested methods include taking the mean of their genes, taking a random point linearly between the two individuals, and taking a random point linearly that could lie outside the two individuals. This implementation instead generates a random child using a Gaussian distribution with mean equal to the mean of the parents' genes, and standard deviation equal to the distance between the parents and the mean. By the Gaussian rule of thumb, this implies that about 68% of children will fall near the mean and between the two parents, while 32% of children will fall outside of the two parents. This promotes convergence while still allowing the population to search nearby locations.

Finally, this implementation uses a modified convergence criteria referred to in [6] as Population Variance. If the standard deviation of the most fit half of the population is less than the "tol" parameter, the algorithm stops. Therefore, as long as the mutation rate is less than 0.5, and the mutation rate is less than the kill rate, the algorithm should converge (intuitively, not mathematically) given enough generations.

The algorithm was tested on various functions five times each and the results recorded. Each test was performed using a population of size 1000, 10000 max iterations, a kill rate of 0.5,

Table 1. Results of genetic algorithm for various test functions (5 simulations each).

Function	Converged	Global	Min Gen.	Mean Gen.	Max Gen.	Min Err.	Mean Err.	Max Err.
ackley	5 / 5	5 / 5	91	92.4	94	0	0	0
sphere	5 / 5	5 / 5	132	132.8	134	0	0	0
rosenbrock	0 / 5	4 / 5	10000	10000	10000	0	5.28	26.4
beale	5 / 5	5 / 5	152	175	249	0	0	0
goldstein	0 / 5	5 / 5	10000	10000	10000	0	0	0
booth	5 / 5	5 / 5	108	110.2	112	0	0	0
bukin6	5 / 5	1 / 5	444	480.4	512	0.0018	0.0078	0.0138
matyas	5 / 5	5 / 5	129	135.2	144	0	0	0
levi13	5 / 5	5 / 5	98	99	101	0	0	0
threehump	5 / 5	5 / 5	94	95.6	97	0	0	0
easom	0 / 5	5 / 5	10000	10000	10000	0	0	0
crosstray	0 / 5	5 / 5	10000	10000	10000	0	0	0
eggholder	0 / 5	5 / 5	10000	10000	10000	0	0	0
holdertable	0 / 5	5 / 5	10000	10000	10000	0	0	0
mccormick	0 / 5	5 / 5	10000	10000	10000	0	0	0
schaffer2	0 / 5	5 / 5	10000	10000	10000	0	0	0
schaffer4	0 / 5	0 / 5	10000	10000	10000	0.208	0.208	0.208
styblinski	0 / 5	5 / 5	10000	10000	10000	0	0	0
simionescu	0 / 5	5 / 5	10000	10000	10000	0	0	0

a mutation rate of 0.5, and a tolerance of 10^{-12}. These tests were carried out using "poptest.py" by changing the value of the global variable "FUNCTION" to any of the values in the array "VALID". The "poptest.py" script prints the current best individual every 10 generations, and terminates after 10000 generations. At the end, it prints the time elapsed over the simulation, the best individual in the population, and the true global minimum.

IV. Results

The results of the simulations are shown above in Table 1. Each function was simulated five times. Only 8/19 of the functions met the convergence criteria and terminated before reaching the full 10000 generations. On the other hand, the algorithm found the global minimum every time for 16/19 of the functions. The "schaffer4" function was the only function for which the algorithm never found the global optimum. Images and definitions of the functions can be found in Appendix B.

V. Discussion and Conclusions

The convergence criteria used in the algorithm did not work well in practice, since 11/19 of the functions did not converge. Moreover, 10/11 of the functions that did not converge did find the global minimum. In fact, for those 10 functions, in most cases the algorithm found the global minimum (or at least a point very close to the global minimum) in only a few hundred generations. The rest of the algorithm was likely attempting to descend a hill, which is notoriously difficult for genetic algorithms. The performance would likely improve significantly by increasing the "tol" parameter, finding an estimate of the global minimum, and then using a gradient-descent method to gain precision.

Overall I think these results are pretty cool, since none of the parameters of the algorithm were changed from function to function. This implies that (at least in low dimensional problems) genetic algorithms are somewhat independent of their parameters. Furthermore, for those functions that converged, the number of iterations was fairly constant. This likely indicates that genetic algorithms are at least somewhat independent of the random initial population.

As mentioned in [3], it is obvious that genetic algorithms are not immune to getting stuck in local minima. Although it certainly did better than a gradient-descent method would have, it still got stuck in a local minimum at least once for 3/19 functions. It failed to find the global minimum even once for the "schaffer4" function.

Although, these conclusions should be taken with a grain of salt, since 5 simulations per function is nowhere near enough to characterize performance with any statistical certainty.

Acknowledgments

I would like to thank Dr. Bouaynaya for teaching Intro to Optimization, as I think it is a topic every engineer should learn about and explore.

I would like to acknowledge Dr. Xin Yao for kindling my interest in evolutionary computation at the IEEE

Computational Intelligence Society Workshop at the University of Rhode Island.

Finally, the pictures in Appendix B are courtesy of Wikipedia under the Creative Commons License.

REFERENCES

[1] N. Bouaynaya. Class Lecture. Topic: "Lecture #8: Gradient Methods." Faculty of Electrical and Computer Engineering, Rowan University, Glassboro, NJ, Oct. 9, 2014.

[2] W. Lin et al. "Adapting Crossover and Mutation Rates in Genetic Algorithms." *Journal of Information Science and Engineering*, vol. 19, 2003, pg. 889-903.

[3] O. Hrstka and A. Kucerova. "Improvements of real coded genetic algorithms based on differential operators preventing permature convergence." *Advances in Engineering Sofware*, vol. 35, 2004, pg. 237-246.

[4] M. Safe et al. "On Stopping Criteria for Genetic Algorithms." *Advances in Artificial Intelligence*, 2004.

[5] D. Thierens. "Selection Schemes, Elitist Recombination, and Selection Intensity." *International Computer Games Association Journal*, 1997, pg. 152-159.

[6] D. Bhandari et al. "Variance as a Stopping Criterion for Genetic Algorithms with Elitist Model." *Fundamenta Informaticae*, vol. 120, pg. 145-164.

[7] S. Mashohor et al. "Elitist Selection Schemes for Genetic Algorithm based Printed Circuit Board Inspection System." *IEEE Congress on Evolutionary Computation*, vol. 2, 2005, pg. 974-978.

[8] B. Tomoiaga et al. "Pareto Optimal Reconfiguration of Power Distribution Systems Using a Genetic Algorithm Based on NSGA-II." *Energies*, vol.6, 2013, pg. 1439-1455.

[9] V. Ojha et al. "Application of Real Valued Neuro Genetic Algorithm in Detection of Components Present in Manhole Gas Mixture." *Advances in Intelligent and Soft Computing*, vol. 166, 2012, pg. 333-340.

[10] R. Oldenhuis. "Test functions for global optimization algorithms." *Mathworks File Exchange*, 2014.

[11] E. Jones et al. "SciPy: Open Source Scientific Tools for Python." Online. 2001. http://www.scipy.org/

Appendix A.1 – population.py

```
#!/usr/bin/env python3

import time,random
import numpy as np

class Population:

  def __init__(self,fit,con,mins,maxs,
      pop=None,pop_size=1000,kill_rate=0.5,mut_rate=0.25):
    """Create a new population for genetic optimization"""

    # Set required args
    self.fit_fun = fit
    self.con_fun = con
    self.mins = mins
    self.maxs = maxs

    # Set optional args
    self.pop = pop
    self.pop_size = pop_size
    self.kill_rate = kill_rate
    self.mut_rate = mut_rate

    # Other setup
    self.genes = len(mins)

    if self.pop is None:
      self.pop = np.zeros((self.pop_size,self.genes))

    self.fitness = np.zeros(self.pop_size)

    self.checkparams()

    random.seed(time.time())

  def checkparams(self):
    """test all parameters for validity"""

    # Type assertions
    assert callable(self.fit_fun), 'Parameter fit must be function'
    assert callable(self.con_fun), 'Parameter con must be function'
    assert isinstance(self.mins,np.ndarray), 'Parameter mins must be numpy.ndarray'
    assert isinstance(self.maxs,np.ndarray), 'Parameter maxs must be numpy.ndarray'
    assert isinstance(self.pop,np.ndarray), 'Parameter pop must be numpy.ndarray'
    assert isinstance(self.pop_size,int), 'Parameter pop_size must be int'
    assert isinstance(self.kill_rate,(int,float)), 'Parameter kill_rate must be int or float'
    assert isinstance(self.mut_rate,(int,float)), 'Parameter mut_rate must be int or float'

    # Individual assertions
    assert len(self.mins.shape)==1, 'Parameter mins must be row vector'
    assert len(self.maxs.shape)==1, 'Parameter maxs must be row vector'
    assert len(self.pop)==self.pop_size, 'Parameter pop must be of length pop_size'
    assert self.pop_size>0, 'Parameter pop_size must be greater than 0'
    assert (self.kill_rate>=0) and (self.kill_rate<1), 'Parameter kill_rate must be in [0,1)'
    assert (self.mut_rate>=0) and (self.mut_rate<1), 'Parameter mut_rate must be in [0,1)'

    # Dependant assertions
    assert self.pop.shape[0]==self.pop_size, 'Parameter pop must have pop_size rows'
    assert self.pop.shape[1]==self.genes, 'Parameter pop must have genes cols'
    assert self.mins.shape==self.maxs.shape, 'Parameters mins and maxs must be same shape'
    assert all(self.maxs>self.mins), 'Parameter maxs must be greater than mins'
```

```
def randpop(self):
  """generate a random population"""

  for r in xrange(0,self.pop_size):
    self.pop[r,:] = self.getrandmem()
  self.updatefit()

def getrandmem(self):
  """generate a random member using uniform dist"""

  feas = False
  mem = np.zeros(self.genes)
  while not feas:
    for g in xrange(0,self.genes):
      mem[g] = random.uniform(self.mins[g],self.maxs[g])
    feas = self.isfeasible(mem)
  return mem

def isfeasible(self,mem):
  """check if a member is feasible"""

  return ((self.con_fun(mem)) and
    all(mem>self.mins) and
    all(mem<self.maxs))

def updatefit(self,ind=None):
  """update fitness values"""

  if ind is None:
    ind = xrange(0,self.pop_size)
  for i in ind:
    self.fitness[i] = self.fit_fun(self.pop[i,:])

def evolve(self,maxiter=1000,converge=True,tol=1e-6):
  """evolve the population until convergence or maxiter"""

  converged = False
  i = 0
  while ((not converged) or (not converge)) and (i<maxiter):
    i += 1
    dead = self.getkilled()
    self.breedpop(dead)
    self.updatefit(dead)
    if converge:
      converged = self.isconverged(tol)
  return i

def getkilled(self):
  """return indices of randomly killed"""

  killnum = int(self.kill_rate*self.pop_size)
  return self.fitness.argsort()[-killnum:]

def getunfit(self,pcnt):
  """return the indices of the most unfit"""

  unfit = int(self.pop_size*pcnt)
  return self.fitness.argsort()[-unfit:]

def getfit(self,pcnt):
  """return the indices of the most fit"""

  fit = int(self.pop_size*pcnt)
  return self.fitness.argsort()[:-(fit+1)]
```

```
def breedpop(self,dead):
  """breed new members replacing the given indices"""

  alive = np.setdiff1d(xrange(0,self.pop_size),dead)
  for d in dead:
    self.pop[d] = self.breedmem(alive)

def breedmem(self,alive):
  """breed a new member from those that are alive"""

  parents = random.sample(alive,2)
  p1 = self.pop[parents[0]]
  p2 = self.pop[parents[1]]
  mu = (p1+p2)/2
  sigma = abs(p2-mu)
  mem = np.zeros(self.genes)
  feas = False
  while not feas:
    for g in xrange(0,self.genes):
      if random.uniform(0,1)<self.mut_rate:
        mem[g] = random.uniform(self.mins[g],self.maxs[g])
      else:
        mem[g] = random.gauss(mu[g],sigma[g])
    feas = self.isfeasible(mem)
  return mem

def isconverged(self,tol):
  """check for convegence"""

  fit = self.pop[self.getfit(0.5)]
  sd = np.std(fit,axis=0)
  return all(sd<tol)

def getmin(self):
  """return the best member as (variables,cost)"""

  minp = self.fitness.argsort()[0]
  return (self.pop[minp],self.fitness[minp])
```

Appendix A.2 – poptest.py

```
#!/usr/bin/env python

import math,time
import numpy as np
from population import Population

VALID = (['ackley','sphere','rosenbrock','beale','goldstein',
          'booth','bukin6','matyas','levi13','threehump',
          'easom','crosstray','eggholder','holdertable','mccormick',
          'schaffer2','schaffer4','styblinski','simionescu'])

FUNCTION = 'holdertable'

def main():
  """run the algorithm on the selected FUNCTION"""

  # get the parameters for the chosen FUNCTION
  (fitness,cons,mins,maxs,globmin) = getfunc(FUNCTION)

  # setup the Population
  p = Population(fitness,cons,mins,maxs,pop_size=1000)
  p.randpop()

  # keep track of iterations and time the algorithm
  iters = 0
  t1 = time.time()

  # run for up to 10000 generations, or until convergence
  # print current best every 10 generations
  for i in xrange(0,int(1e3)):
    add = p.evolve(maxiter=10,tol=1e-12)
    iters += add
    m = p.getmin()
    x1 = m[0][0]
    x2 = m[0][1]
    y = m[1]
    print str(iters).zfill(5)+': ['+str(x1)+','+str(x2)+'] = '+str(y)
    if add<10:
      break

  # print final results
  print '\n'+str(iters)+' generations in '+str(time.time()-t1)+' secs'
  print '\nCalced: ['+str(round(x1,6))+','+str(round(x2,6))+'] = '+str(round(y,6))
  x1 = globmin[0][0]
  x2 = globmin[0][1]
  y = globmin[1]
  print 'Actual: ['+str(round(x1,6))+','+str(round(x2,6))+'] = '+str(round(y,6))

def getfunc(fun):
  """return the cost function, constraint function, mins, maxs, and
  globalmin information for the given string fun or raise an error
  if the given string is invalid"""

  if fun=='ackley':
    fun = ackley
    cons = nocons
    mins = np.array([-5,-5])
    maxs = np.array([5,5])
    globmin = (np.array([0,0]),0)
  elif fun=='sphere':
    fun = sphere
    cons = nocons
    mins = np.array([-1000000,-1000000])
```

```
  maxs = np.array([1000000,1000000])
  globmin = (np.array([0,0]),0)
elif fun=='rosenbrock':
  fun = rosenbrock
  cons = nocons
  mins = np.array([-1000000,-1000000])
  maxs = np.array([1000000,1000000])
  globmin = (np.array([1,1]),0)
elif fun=='beale':
  fun = beale
  cons = nocons
  mins = np.array([-4.5,-4.5])
  maxs = np.array([4.5,4.5])
  globmin = (np.array([3,0.5]),0)
elif fun=='goldstein':
  fun = goldstein
  cons = nocons
  mins = np.array([-2,-2])
  maxs = np.array([2,2])
  globmin = (np.array([0,-1]),3)
elif fun=='booth':
  fun = booth
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
  globmin = (np.array([1,3]),0)
elif fun=='bukin6':
  fun = bukin6
  cons = nocons
  mins = np.array([-15,-3])
  maxs = np.array([-5,3])
  globmin = (np.array([-10,1]),0)
elif fun=='matyas':
  fun = matyas
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
  globmin = (np.array([0,0]),0)
elif fun=='levi13':
  fun = levi13
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
  globmin = (np.array([1,1]),0)
elif fun=='threehump':
  fun = threehump
  cons = nocons
  mins = np.array([-5,-5])
  maxs = np.array([5,5])
  globmin = (np.array([0,0]),0)
elif fun=='easom':
  fun = easom
  cons = nocons
  mins = np.array([-100,-100])
  maxs = np.array([100,100])
  globmin = (np.array([math.pi,math.pi]),-1)
elif fun=='crosstray':
  fun = crosstray
  cons = nocons
  mins = np.array([-10,-10])
  maxs = np.array([10,10])
  globmin = (np.array([1.34941,1.34941]),-2.06261)
elif fun=='eggholder':
  fun = eggholder
  cons = nocons
```

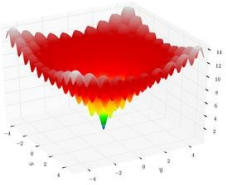
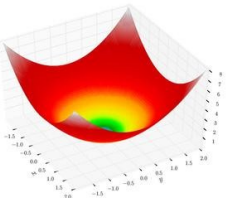
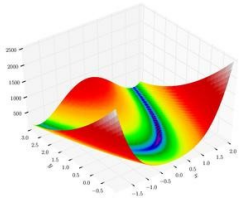
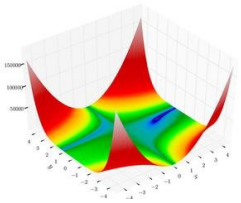
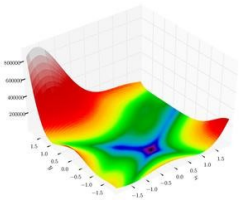
```
    mins = np.array([-512,-512])
    maxs = np.array([512,512])
    globmin = (np.array([512,404.2319]),-959.6407)
  elif fun=='holdertable':
    fun = holdertable
    cons = nocons
    mins = np.array([-10,-10])
    maxs = np.array([10,10])
    globmin = (np.array([8.05502,9.66459]),-19.2085)
  elif fun=='mccormick':
    fun = mccormick
    cons = nocons
    mins = np.array([-1.5,-3])
    maxs = np.array([4,4])
    globmin = (np.array([-0.54719,-1.54719]),-1.9133)
  elif fun=='schaffer2':
    fun = schaffer2
    cons = nocons
    mins = np.array([-100,-100])
    maxs = np.array([100,100])
    globmin = (np.array([0,0]),0)
  elif fun=='schaffer4':
    fun = schaffer4
    cons = nocons
    mins = np.array([-100,-100])
    maxs = np.array([100,100])
    globmin = (np.array([0,1.25313]),0.292579)
  elif fun=='styblinski':
    fun = styblinski
    cons = nocons
    mins = np.array([-5,-5])
    maxs = np.array([5,5])
    globmin = (np.array([-2.903534,-2.903534]),-78.33198)
  elif fun=='simionescu':
    fun = simionescu
    cons = simioinescucons
    mins = np.array([-1.25,-1.25])
    maxs = np.array([1.25,1.25])
    globmin = (np.array([0.84852813,-0.84852813]),-0.072)
  else:
    raise ValueError('Function "'+str(fun)+'" is not a valid function')
  return (fun,cons,mins,maxs,globmin)

def ackley(genes):
  """cost function for the ackley function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-20*math.exp(-0.2*pow(0.5*(x1**2+x2**2),0.5))-
    math.exp(0.5*(math.cos(2*math.pi*x1)+math.cos(2*math.pi*x2)))+
    math.e+20)

def sphere(genes):
  """cost function for the sphere function"""

  x1 = genes[0]
  x2 = genes[1]
  return x1**2+x2**2

def rosenbrock(genes):
  """cost function for the rosenbrock function"""

  x1 = genes[0]
  x2 = genes[1]
  return (100*pow(x2-x1**2,2)+pow(x1-1,2))
```

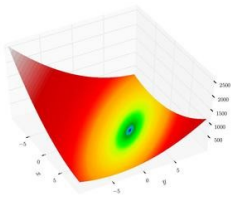
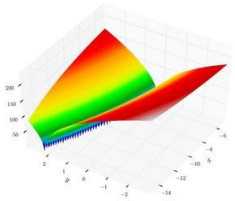
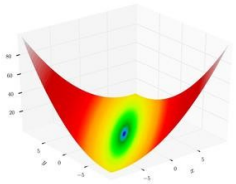
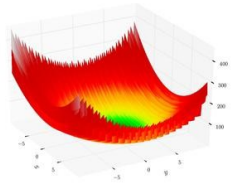
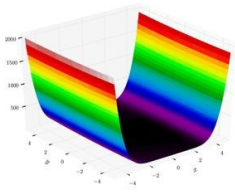
```
def beale(genes):
  """cost function for the beale function"""

  x1 = genes[0]
  x2 = genes[1]
  return (pow(1.5-x1+x1*x2,2)+
    pow(2.25-x1+x1*pow(x2,2),2)+
    pow(2.625-x1+x1*pow(x2,3),2))

def goldstein(genes):
  """cost function for the goldstein function"""

  x1 = genes[0]
  x2 = genes[1]
  return ((1+pow(x1+x2+1,2)*(19-14*x1+3*x1**2-14*x2+6*x1*x2+3*x2**2))*
    (30+pow(2*x1-3*x2,2)*(18-32*x1+12*x1**2+48*x2-36*x1*x2+27*x2**2)))

def booth(genes):
  """cost function for the booth function"""

  x1 = genes[0]
  x2 = genes[1]
  return (pow(x1+2*x2-7,2)+pow(2*x1+x2-5,2))

def bukin6(genes):
  """cost function for the bukin6 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (100*pow(abs(x2-0.01*x1**2),0.5)+0.01*abs(x1+10))

def matyas(genes):
  """cost function for the matyas function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.26*(x1**2+x2**2)-0.48*x1*x2)

def levi13(genes):
  """cost function for the levi13 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (math.sin(3*math.pi*x1)**2+((x1-1)**2)*
    (1+math.sin(3*math.pi*x2)**2)+((x2-1)**2)*
    (1+math.sin(2*math.pi*x2)**2))

def threehump(genes):
  """cost function for the threehump function"""

  x1 = genes[0]
  x2 = genes[1]
  return (2*x1**2-1.05*x1**4+(x1**6)/6+x1*x2+x2**2)

def easom(genes):
  """cost function for the easom function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-math.cos(x1)*math.cos(x2)*
    math.exp(-((x1-math.pi)**2+(x2-math.pi)**2)))
```

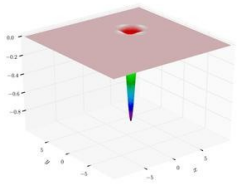
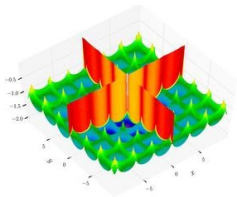
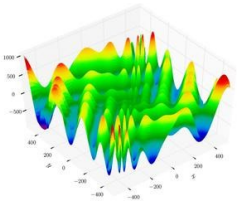
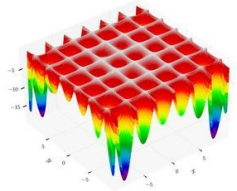
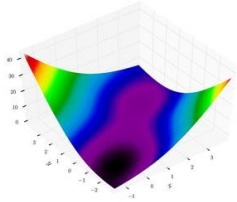
```
def crosstray(genes):
  """cost function for the crosstray function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-0.0001*(abs(math.sin(x1)*math.sin(x2)*
    math.exp(abs(100-math.sqrt(x1**2+x2**2)/math.pi)))+1)**0.1)

def eggholder(genes):
  """cost function for the eggholder function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-(x2+47)*math.sin(math.sqrt(abs(x2+x1/2+47)))-
    x1*math.sin(math.sqrt(abs(x1-(x2+47)))))

def holdertable(genes):
  """cost function for the holdertable function"""

  x1 = genes[0]
  x2 = genes[1]
  return (-abs(math.sin(x1)*math.cos(x2)*
    math.exp(abs(1-math.sqrt(x1**2+x2**2)/math.pi))))

def mccormick(genes):
  """cost function for the mccormick function"""

  x1 = genes[0]
  x2 = genes[1]
  return (math.sin(x1+x2)+(x1-x2)**2-1.5*x1+2.5*x2+1)

def schaffer2(genes):
  """cost function for the schaffer2 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.5+((math.sin(x1**2-x2**2)**2-0.5)/
    (1+0.001*(x1**2+x2**2))**2))

def schaffer4(genes):
  """cost function for the schaffer4 function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.5+((math.cos(math.sin(abs(x1**2-x2**2)))-0.5)/
    (1+0.001*(x1**2+x2**2))**2))

def styblinski(genes):
  """cost function for the styblinski function"""

  x1 = genes[0]
  x2 = genes[1]
  return (((x1**4-16*x1**2+5*x1)+(x2**4-16*x2**2+5*x2))/2)

def simionescu(genes):
  """cost function for the simionescu function"""

  x1 = genes[0]
  x2 = genes[1]
  return (0.1*x1*x2)
```

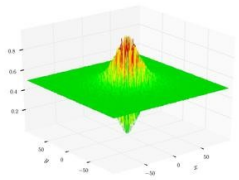
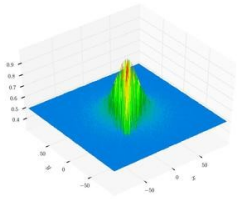
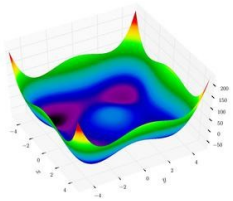
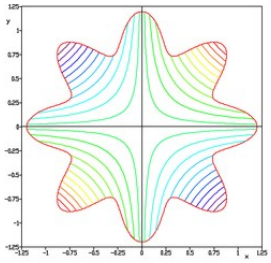
```
def simioinescucons(genes):
  """constraint function for the simionescu function"""

  x1 = genes[0]
  x2 = genes[1]
  return x1**2+x2**2<=(1+0.2*math.cos(2*math.atan(x1/x2)))**2

def nocons(genes):
  """constraint function for functions with only mins and maxs"""

  return True

if __name__ == '__main__':
  main()
```

Appendix B – Test Functions

Function	Formula	Minimum
 Ackley	$f(x,y) = -20\exp\left(-0.2\sqrt{0.5\left(x^2+y^2\right)}\right)$ $-\exp\left(0.5\left(\cos\left(2\pi x\right)+\cos\left(2\pi y\right)\right)\right)+e+20$	$f(0,0) = 0$ $-5 \le (x,y) \le 5$
 Sphere	$f(x,y) = x^2+y^2$	$f(0,0) = 0$ $-\infty \le (x,y) \le \infty$
 Rosenbrock	$f(x,y) = 100\left(y-x^2\right)^2+(x-1)^2$	$f(1,1) = 0$ $-\infty \le (x,y) \le \infty$
 Beale	$f(x,y) = (1.5-x+xy)^2+(2.25-x+xy^2)^2$ $+(2.625-x+xy^3)^2$	$f(3,0.5) = 0$ $-4.5 \le (x,y) \le 4.5$
 Goldstein-Price	$f(x,y) = \left(1+(x+y+1)^2\left(19-14x+3x^2-14y+6xy+3y^2\right)\right)$ $\left(30+(2x-3y)^2\left(18-32x+12x^2+48y-36xy+27y^2\right)\right)$	$f(0,-1) = 3$ $-2 \le (x,y) \le 2$

 Booth	$f(x,y) = (x+2y+7)^2+(2x+y-5)^2$	$f(1,3) = 0$ $-10 \le (x,y) \le 10$
 Bukin N.6	$f(x,y) = 100\sqrt{\lvert y-0.01x^2\rvert}+0.01\lvert x+10\rvert$	$f(-10,1) = 0$ $-15 \le x \le -5$ $-3 \le y \le 3$
 Matyas	$f(x,y) = 0.26(x^2+y^2)-0.48xy$	$f(0,0) = 0$ $-10 \le (x,y) \le 10$
 Lévi N.13	$f(x,y) = \sin^2(3\pi x)+(x-1)^2(1+\sin^2(3\pi y))$ $+(y-1)^2(1+\sin^2(2\pi y))$	$f(1,1) = 0$ $-10 \le (x,y) \le 10$
 Three-hump camel	$f(x,y) = 2x^2-1.05x^4+\frac{x^6}{6}+xy+y^2$	$f(0,0) = 0$ $-5 \le (x,y) \le 5$

 Easom	$f(x,y) = -\cos(x)\cos(y)\exp(-((x-\pi)^2+(y-\pi)^2))$	$f(\pi,\pi) = -1$ $-100 \le (x,y) \le 100$
 Cross-in-tray	$f(x,y) = -0.0001\left(\left\lvert \sin(x)\sin(y)\exp\left(\left\lvert 100-\frac{\sqrt{x^2+y^2}}{\pi}\right\rvert\right)\right\rvert+1\right)^{0.1}$	$f(1.349,-1.349)$ $f(1.349,1.349)$ $f(-1.349,1.349)$ $f(-1.349,-1.349)$ $= -2.06261$ $-10 \le (x,y) \le 10$
 Eggholder	$f(x,y) = -(y+47)\sin\left(\sqrt{\left\lvert y+\frac{x}{2}+47\right\rvert}\right) -x\sin\left(\sqrt{\left\lvert x-(y+47)\right\rvert}\right)$	$f(512,404.23)$ $= -959.6407$ $-512 \le (x,y) \le 512$
 Hölder table	$f(x,y) = -\left\lvert \sin(x)\cos(y)\exp\left(\left\lvert 1-\frac{\sqrt{x^2+y^2}}{\pi}\right\rvert\right)\right\rvert$	$f(8.055,-9.665)$ $f(8.055,9.665)$ $f(-8.055,9.665)$ $f(-8.055,-9.665)$ $= -19.2085$ $-10 \le (x,y) \le 10$
 McCormick	$f(x,y) = \sin(x+y)+(x-y)^2-1.5x+2.5y+1$	$f(-0.547,-1.547)$ $= -1.9133$ $-1.5 \le x \le 4$ $-3 \le y \le 4$

 Schaffer N.2	$f(x,y) = 0.5+\frac{\sin^2(x^2-y^2)-0.5}{(1+0.001(x^2+y^2))^2}$	$f(0,0) = 0$ $-100 \le (x,y) \le 100$
 Schaffer N.4	$f(x,y) = 0.5+\frac{\cos(\sin(\lvert x^2-y^2\rvert))-0.5}{(1+0.001(x^2+y^2))^2}$	$f(0,1.25313) = 0.292579$ $-100 \le (x,y) \le 100$
 Styblinski-Tang	$f(x,y) = 0.5((x^4-16x^2+5x)+(y^4-16y^2+5y))$	$f(-2.904,-2.904) = -78.33198$ $-5 \le (x,y) \le 5$
 Simionescu	$f(x,y) = 0.1xy$ subject to: $x^2+y^2 \le \left(1+0.2\cos\left(8\arctan\frac{x}{y}\right)\right)^2$	$f(0.849,-0.849)$ $f(-0.849,0.849) = -0.072$ $-1.25 \le (x,y) \le 1.25$